International Journal of Emerging Technology and Innovative Engineering
Volume I, Issue7, July 2015 (ISSN: 2394 – 6598)

ANALYSIS OF OBJECT ORIENTED SOFTWARE METRICS

Pooja Arora

Assistant Professor, BCIIT(affiliated to GGSIPU), Kalkaji,Delhi.

pooja@bciit.ac.in

ABSTRACT:
Object oriented analysis and design is becoming more popular in software development environment and object oriented analysis and design metrics is an essential part of software environment. This study focus on a set of object oriented metrics that can be used to measure the quality of an object oriented analysis and design.
A metrics-based means to both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance .Measuring these 3 aspects at system level provides a comprehensive characterization of an entire system
1. Inheritance
2. size and complexity
3. coupling
In most object-oriented languages we find and can measure classes; operations (including methods and functions); variables (including the whole range from attributes to local variables) etc. We may want to assess the size, the complexity, the quality, etc.
The metrics for object oriented design focus on measurements that are applied to the class and design characteristics. These measurements permit designers to access the software early in process, making changes that will reduce complexity and improve the continuing capability of the design.
This report summarizes the existing metrics, which will guide the analyst and designers to support their analysis and design. We have categorized metrics and discussed in such a way that novice analyst and designers can apply metrics in their respective work as needed**.**

KEYWORDS: Metrics; complexity; Software engineering metrics; Object oriented Software engineering metrics; Localization; encapsulation; information hiding; inheritance; abstraction

I. INTRODUCTION

METRICS FOR ANALYSIS : When code is analyzed for object-oriented metrics, often two suites of metrics are used, the Chidamber-Kemerer (CK) [8] and MOOD [1, 2] suites. In this section, we enumerate and explain the specific measures that can be computed using this tool.

1.1 Coupling

In 1974, Stevens et al. first defined coupling in the context of structured development as "the measure of the strength of association established by a connection from one module to another [21]."Coupling is a measure of interdependence of two objects. For example, objects A and B are coupled if a method of object A calls a method or accesses a variable in object B. Classes are coupled when methods declared in one class use methods or attributes of the other classes.

The *Coupling Factor (CF)* is evaluated as a fraction. The numerator represents the number of non-inheritance couplings. The denominator is the maximum number of couplings in a system. The maximum number of couplings includes both inheritance and non-inheritance related coupling. Inheritance-based couplings arise as derived classes (subclasses) inherit methods and attributes form its base class (superclass). The CF metric is included in the MOOD metric suite.

Empirical evidence supports the benefits of low coupling between objects [6, 7, 20]. The main arguments are that the stronger the coupling between software artifacts,

(i) the more difficult it is to understand individual artifacts, and hence to correctly maintain or enhance them;

(ii) the larger the sensitivity of (unexpected) change and defect propagation effects across artifacts; and

(iii) consequently, the more testing required to achieve satisfactory reliability levels. Additionally, excessive coupling between objects is detrimental to modular design and prevents reuse. To summarize, low coupling is desirable.

1.2 Cohesion

Cohesion refers to how closely the operations in a class are related to each other. Cohesion of a class is the degree to which the local methods are related to the local instance variables in the class. The CK metrics suite examines the Lack of Cohesion (LOCOM), which is the number of disjoint/non-intersection sets of local methods [12].

There are at least two different ways of measuring cohesion:

Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.

Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates the design implementation as well as reusability.

1.3 Encapsulation

Information hiding is a way of designing routines such that only a subset of the module's properties, its public interface, are known to users of the module. Information hiding gives rise to encapsulation in object-oriented languages. "Encapsulation means that all that is seen of an object is its interface, namely the operations we can perform on the object [17]."Information hiding is a theoretical technique that indisputably proven its value in practice. "Large programs that use information hiding have been found to be easier to modify -- by a factor of 4 -- than programs that don't [4, 18] ."

The following two encapsulation measures are contained in the MOOD metrics suite.

1.3.1 Attribute Hiding Factor (AHF)

The Attribute Hiding Factor measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible. An attribute is called visible if it can be accessed by another class or object. Attributes should be "hidden" within a class. They can be kept from being accessed by other objects by being declared a private.

The Attribute Hiding Factor is a fraction. The numerator is the sum of the invisibilities of all attributes defined in all classes. The denominator is the total number of attributes defined in the project [10]. It is desirable for the Attribute Hiding Factor to have a large value.

1.3.2 Method Hiding Factor (MHF)

The Method Hiding Factor measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.

The Method Hiding Factor is a fraction where the numerator is the sum of the invisibilities of all methods defined in all classes. The denominator is the total number of methods defined in the project.

Methods should be encapsulated (hidden) within a class and not available for use to other objects. Method hiding increases reusability in other applications and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden. Thus hiding methods also reduces modifications to the code [10]. The Method Hiding Factor should have a large value.

1.4 Inheritance

Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

1.4.1 Depth of Inheritance Tree (DIT)

The depth of a class within the inheritance hierarchy is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritance, the DIT is the maximum length from the node to the root of the tree [8].

Well structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour and, therefore, more fault-prone [15]. Deeper trees require greater design complexity, since more methods and classes are involved [8]. Indeed, deep hierarchies are also a conceptual integrity concern because it becomes difficult to determine which class to specialize from [3]. Additionally, interface changes within the tree must be reflected throughout the entire class tree and object instances. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods [8].

Applications can be considered to be "top heavy" if there are too many classes near the root, and indication that designers may not be taking advantage of reuse of methods through inheritance. Alternatively, applications can be considered to be "bottom heavy" whereby too many classes are near the bottom of the hierarchy, resulting in concerns related to design complexity and conceptual integrity.

1.4.2 Number of Children (NOC)

This metric is the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be difficult to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and therefore must be more flexible [3].

1.5 Complexity

1.5.1 Weighted Methods/Class (WMC)

WCM measures the complexity of an individual class. A class with more member functions than its peers is considered to be more complex and therefore more error prone [3]. The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This reasoning indicates that a smaller number of methods is good for usability and reusability.

However, more recently, software development trends support have more, smaller methods over fewer, larger methods for reduced complexity, increased readability, and improved understanding [13]. This reasoning contradicts the prior reasoning of the last paragraph. The most recent recommendation (of more, smaller methods) is most likely more prudent. However, if a method is in a large inheritance tree having a large number of methods may not be advisable.

1.6 Shyam Chidamer and Chris Kemerer have developed a small metrics suite for object-oriented designs. The six metrics they have identified are:

weighted methods per class: This focuses on the complexity and number of methods within a class.

depth of inheritance tree: This is a measure of how many layers of inheritance make up a given class hierarchy.

number of children: This is the number of immediate specializations for a given class.

coupling between object classes: This is a count of the number of other classes to which a given class is coupled.

response for a class: This is the size of the set of methods that can potentially be executed in response to a message received by an object.

lack of cohesion in methods: This is a measure of the number of different methods within a class that reference a given instance variable.

1.7 Summary of Metrics

The table below summarizes the metrics discussed above. It illustrates, in general, whether a high or low value is desired from a metric for better code quality. However, one still must exercise judgment when determining the best approach for the task at hand.

Metric	Desirable Value
Coupling Factor	Lower
Lack of Cohesion of Methods	Lower
Cyclomatic Complexity	Lower
Attribute Hiding Factor	Higher
Method Hiding Factor	Higher
Depth of Inheritance Tree	Low (tradeoff)
Number of Children	Low (tradeoff)
Weighted Methods Per Class	Low (tradeoff)
Number of Classes	Higher
Lines of Code	Lower

Table 1: Summary of Metrics

II. ANALYSIS OF EXISTING METRICS

2.1. Chidamber & Kemerer's Metrics Suite

Chidamber and Kemerer's metrics suite for OO Design is the deepest research in OO metrics investigation. They have defined six metrics for the OO design. In this section we'll have a complete description of their metrics:

Metric 1: Weighted Methods per Class (WMC)

Definition: Consider a Class C1, with methods M1... Mn that are defined in the class. Let c_1... c_n be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

If all method complexities are considered to be unity, then WMC = n, the number of methods.

Theoretical basis: WMC relates directly to Bunge's definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties. The number of methods is, therefore, a measure of class definition as well as being attributes of a class, since attributes correspond to properties.

Viewpoints

• The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.

• The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.

• Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Metric 2: Depth of Inheritance Tree (DIT)

Definition: Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

Theoretical basis: DIT relates to Bunge's notion of the scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class.

Viewpoints:

• The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour.

• Deeper trees constitute greater design complexity, since more methods and classes are involved.

• The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Metric 3: Number of children (NOC)

Definition: NOC = number of immediate sub-classes subordinated to a class in the class hierarchy.

Theoretical basis: NOC relates to the notion of scope of properties. It is a measure of how many subclasses are going to inherit the methods of the parent class.

Viewpoints:

• Greater the number of children, greater the reuse, since inheritance is a form of reuse.

• Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing.

• The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

Metric 4: Coupling between object classes (CBO)

Definition: CBO for a class is a count of the number of other classes to which it is coupled.

Theoretical basis: CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. As stated earlier, since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

Viewpoints:

• Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.

• In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

• A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Metric 5: Response for a Class (RFC)

Definition: RFC = | RS | where RS is the response set for the class.

Theoretical basis: The response set for the class can be expressed as: RS = { M }∪ all i { Ri } where { Ri } = set of methods called by method i and { M } = set of all methods in the class The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. The cardinality of this set is a measure of the attributes of objects in the class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

Viewpoints:

• If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

• The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

• A worst case value for possible responses will assist in appropriate allocation of testing time.

Metric 6: Lack of Cohesion in Methods (LCOM)

Definition: Consider a Class C1 with n methods M1, M2..., Mn. Let {Ij} = set of instance variables used by method Mi. There are n such sets {I1},... {In}. Let P = { (Ii,Ij) | Ii ∩ Ij = ∅ } and Q = { (Ii,Ij) | Ii ∩ Ij ≠ ∅ }. If all n sets {I1},... {In} are ∅ then let P = ∅ . LCOM = |P| - |Q|

Example: Consider a class C with three methods M1, M2 and M3. Let {I1} = {a,b,c,d,e} and {I2} = {a,b,e} and {I3} = {x,y,z}. {I1} ∩ {I2} is non-empty, but {I1} ∩ {I3} and {I2} ∩ {I3} are null sets. LCOM is the (number of null-intersections - number of non-empty intersections), which in this case is 1.

Theoretical basis: This uses the notion of degree of similarity of methods. The degree of similarity for two methods M1 and M2 in class C1 is given by: σ() = {I1} ∩ {I2} where {I1} and {I2} are the sets of instance variables used by M1 and M2

The LCOM is a count of the number of method pairs whose similarity is 0 (i.e. σ() is a null set) minus the count of method pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the interrelatedness between portions of a program. If none of the methods of a class display any instance behaviour, i.e. do not use any instance variables, they have no similarity and the LCOM value for the class will be zero. The LCOM value provides a measure of the relative disparate nature of methods in the class. A smaller number of disjoint pairs (elements of set P) implies greater similarity of methods. LCOM is intimately tied to the instance variables and methods of a class, and therefore is a measure of the attributes of an object class.

Viewpoints:

• Cohesiveness of methods within a class is desirable, since it promotes encapsulation.

• Lack of cohesion implies classes should probably be split into two or more sub-classes.

• Any measure of disparateness of methods helps identify flaws in the design of classes.

• Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

2.2 MOOD (Metrics for Object Oriented Design) The MOOD metrics set refers to a basic structural mechanism of the OO paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphisms (PF) , message-passing (CF) and are expressed as quotients. The set includes the following metrics:

2.2.1 Method Hiding Factor (MHF) MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible. note : inherited methods not considered.

2.2.2 Attribute Hiding Factor (AHF) AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

2.2.3 Method Inheritance Factor (MIF) MIF is defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.

2.2.4 Attribute Inheritance Factor (AIF) AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.

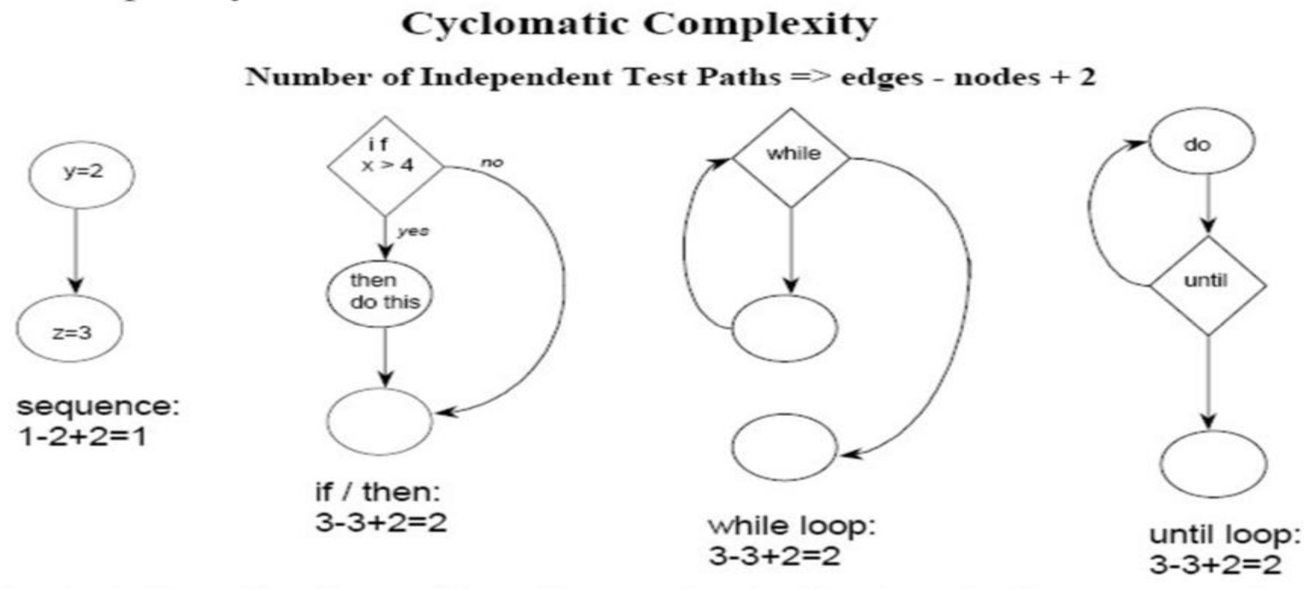
2.2.5 Polymorphism Factor (PF) PF is defined as the ratio of the actual number of possible different polymorphic situation for class Ci to the maximum number of possible distinct polymorphic situations for class Ci.

2.2.6 Coupling Factor (CF) CF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance.

2.3 Some Traditional Metrics There are many metrics that are applied to traditional functional development. The Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center, from experience, has identified three of these metrics that are applicable to object oriented development: Complexity, Size, and Readability. To measure the complexity, the cyclomatic complexity is used.

Metric 1: Cyclomatic Complexity (CC) Cyclomatic complexity (McCabe) is used to evaluate the complexity of an algorithm in a method. It is a count of the number of test cases that are needed to test the method comprehensively. The formula for calculating the cyclomatic complexity is the number of edges minus the number of nodes plus 2. For a sequence where there is only one path, no choices or option, only one test case is needed. An IF loop however, has two choices, if the condition is true, one path is tested; if the condition is false, an alternative path is tested.

Figure 1 shows a method with a low cyclomatic complexity is generally better. This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of Complexity, it also is related to all of the other attributes.



example of calculations for the cyclomatic complexity for four basic programming structure

Figure 1

Metric 2: Size Size of a class is used to evaluate the ease of understanding of code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, the number of blank lines, and the number of comment lines. Lines of Code(LOC) counts all lines. Non-comment Non-blank (NCNB) is sometimes referred to as Source Lines of Code and counts all lines that are not comments and not blanks. Executable Statements (EXEC) is a count of executable statements regardless of number of physical lines of code. For example, in FORTRAN and IF statement may be written: IF X=3 THEN Y=0 This example would be 3 LOC, 3 NCNB, and 1 EXEC. Executable statements is the measure least influenced by programmer or language style. Therefore, since NASA programs are frequently written using multiple languages, the SATC uses executable statements to evaluate project size. Thresholds for evaluating the meaning of size measures vary depending on the coding language used and the complexity of the method. However, since size affects ease of understanding by the developers and maintainers, classes and methods of large size will always pose a higher risk.

Metric 3: Comment Percentage The line counts done to compute the Size metric can be expanded to include a count of the number of comments, both on-line (with code) and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Since comments assist developers and maintainers, higher comment percentages increase understandability and maintainability.

2.4 Complexity Metrics and Models

2.4.1 Halstead's Software Science

The Software Science developed by M.H. Halstead principally attempts to estimate the programming effort. The measurable and countable properties are:

• n1 = number of unique or distinct operators appearing in that implementation • n2 = number of unique or distinct operands appearing in that implementation • N1 = total usage of all of the operators appearing in that implementation

• N2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

I. the vocabulary n as n = n1 + n2

II. the implementation length N as N = N1 + N2 Operators can be "+" and "*" but also an index "[...]" or a statement separation "..;..". The number of operands consists of the numbers of literal expressions, constants and variables.

2.4.2 Length Equation

It may be necessary to know about the relationship between length N and vocabulary n. Length Equation is as follows. " ' " on N means it is calculated rather than counted : N ' = n1log2n1 + n2log2n2 It is experimentally observed that N ' gives a rather close agreement to program length.

2.4.3 Quantification of Intelligence Content

The same algorithm needs more consideration in a low level programming language. It is easier to program in Pascal rather than in assembly. The intelligence Content determines how much is said in a program. In order to find Quantification of Intelligence Content we need some other metrics and formulas:

Program Volume: This metric is for the size of any implementation of any algorithm. V = Nlog2n

Program Level: It is the relationship between Program Volume and Potential Volume. Only the most clear algorithm can have a level of unity. L = V* / V

Program Level Equation: is an approximation of the equation of the Program Level. It is used when the value of Potential Volume is not known because it is possible to measure it from an implementation directly. L ' = n*1n2 / n1N2

Intelligence Content I = L ' x V = (2n2 / n1N2) x (N1 + N2)log2(n1 + n2) In this equation all terms on the right-hand side are directly measurable from any expression of an algorithm. The intelligence content is correlated highly with the potential volume. Consequently, because potential volume is independent of the language, the intelligence content should also be independent.

2.4.3 Programming Effort

The programming effort is restricted to the mental activity required to convert an existing algorithm to an actual implementation in a programming language. In order to find Programming effort we need some metrics and formulas:

Potential Volume: is a metric for denoting the corresponding parameters in an algorithm's shortest possible form. Neither operators nor operands can require repetition. V ' = (n*1 + n*2) log2 (n*1 + n*2)

Effort Equation The total number of elementary mental discriminations is: E = V / L = V2 / V' If we express it: The implementation of any algorithm consists of N selections (nonrandom > of a vocabulary n. a program is generated by making as many mental comparisons as the program volume equation determines, because the program volume V is a measure of it. Another aspect that influences the effort equation is the program difficulty. Each mental comparison consists of a number of elementary mental discriminations. This number is a measure for the program difficulty.

Time Equation A concept concerning the processing rate of the human brain, developed by the psychologist John Stroud, can be used. Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination. The Stroud number S is then Stroud's moments per second with 5 <= S <= 20. Thus we can derive the time equation where, except for the Stroud number S, all of the parameters on the right are directly measurable: T ' = (n1N2(n1log2n1 + n2log2n2) log2n) / 2n2S

2.5 McCabe's Cyclomatic number

A measure of the complexity of a program was developed by McCabe. He developed a system which he called the cyclomatic complexity of a program. This system measures the number of independent paths in a program, thereby placing a numerical value on the complexity. In practice it is a count of the number of test conditions in a program. The cyclomatic complexity (CC) of a graph (G) may be computed according to the following formula: CC(G) = Number (edges) - Number (nodes) + 1 The results of multiple experiments (G.A. Miller) suggest that modules approach zero defects when McCabe's Cyclomatic Complexity is within 7 ± 2. A study of PASCAL and FORTRAN programs (Lind and Vairavan 1989) found that a Cyclomatic Complexity between 10 and 15 minimized the number of module changes.

2.6 Fan-In Fan-Out Complexity - Henry's and Kafura's

Henry and Kafura (1981) identified a form of the fan in - fan out complexity which maintains a count of the number of data flows from a component plus the number of global data structures that the program updates. The data flow count includes updated procedure parameters and procedures called from within a module. Complexity = Length x (Fan-in x Fan-out)2 Length is any measure of length such as lines of code or alternatively McCabe's cyclomatic complexity is sometimes substituted. Henry and Kafura validated their metric using the UNIX system and suggested that the measured complexity of a component allowed potentially faulty system components to be identified. They found that high values of this metric were often measured in components where there had historically been a high number of problems.

2.7 Traditional metrics The metrics presented hereinafter have been selected from the most well known software metrics that have been proposed and could be easily applied to object– oriented programming as well. They are sorted alphabetically according to their codes.

AML (Average Module Length) measures the average module size.

BAM (Binding Among Modules) measures data sharing among modules.

CCN (Cyclomatic Complexity Number) measures the number of decisions in the control graph.

CDF (Control flow complexity and Data Flow complexity) is a combine metric based on variable definitions and cross-references.

COC (Conditions and Operations Count) counts pairs of all conditions and loops within the operations.

COP (Complexity Pair) combines cyclomatic complexity with logic structure.

COR (Coupling Relation) assigns a relation to every couple of modules according to the kind of coupling.

CRM (Cohesion Ratio Metrics) measure the number of modules having functional cohesion divided by the total number of modules.

DEC (Decision Count) offers a method to measure program complexity.

DSI (Delivered Source Instructions) counts separate statements on the same physical line as distinct and ignores comment lines.

ERE (Extent of Reuse) categorises a unit according the lever of reuse (modifications required).

ESM (Equivalent Size Measure) measures the percentage of modifications on a reused module.

EST (Executable Statements) counts separate statements on the same physical line as distinct and ignores comment lines, data declarations and headings.

FCO (Function Count) measures the number of functions and the source lines in every function.

FUP (Function Points) measures the amount of functionality in a system.

GLM (Global Modularity) describes global modularity in terms of several specific views of modularity.

IFL (Information Flow) measures the total level of information flow between individual modules and the rest of a system.

KNM (Knot Measure) is the total number of crossing points on control flow lines.

LOC (Lines Of Code) measures the size of a module.

LVA (Live Variables) deals with the period each variable is used.

MNP (Minimum Number of Paths) measures the minimum number of paths in a program and the reachability of any node.

MOR (Morphology metrics) measure morphological characteristics of a module, such as size, depth, width and edge-to-node ratio.

NLE (Nesting Levels) measures the complexity as depth of nesting.

SSC (composite metric of Software Science and Cyclomatic complexity) combines software science metrics with McCabe's complexity measure.

SSM (Software Science Metrics) are a set of composite size metrics.

SWM (Specification Weight Metrics) measure the function primitives on a given data flow diagram.

TRI (Tree Impurity) determines how far a graph deviates from being a tree .

TRU (Transfer Usage) measures the logical structure of the program.

2.8 Object–Oriented Metrics The metrics presented hereinafter have been selected from metrics proposed specifically for object–oriented measurements and cannot be applied to another programming style. The categories chosen to present the metrics are not defining a metrics classification but used simply to ease the presentation and sometimes a metric may fall in more than one category. The metrics presented are:

class related metrics

method related metrics

inheritance metrics

metrics measure coupling

metrics measure general (system) software production characteristics.

They are sorted alphabetically, according to the codes, as follows.

2.8.1 CLASS METRICS

AHF (Attribute Hiding Factor) is the ratio of the sum of inherited attributes in all system classes under consideration to the total number of available classes attributes.

CCO (Class Cohesion) measures relations between classes.

CEC (Class Entropy Complexity) measures the complexity of classes based on their information content.

CLM (Comment Lines per Method) measures the percentage of comments in methods.

DAM (Data Access Metric) is the ratio of the number of private attributes to the total number of attributes declared in the class.

FOC (Function Oriented Code) measures the percentage of non object–oriented code that is used in a program.

INP (Internal Privacy) refers to the use of accessory functions even within a class.

LCM (Lack of Cohesion between Methods) indicates the level of cohesion between the methods.

MAA (Measure of Attribute Abstraction) is the ratio of the number of attributes inherited by a class to the total number of attributes in the class.

MFA (Measure of Functional Abstraction) is the ratio of the number of methods inherited by a class to the total number of methods accessible by members in the class.

MHF (Method Hiding Factor) is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.

NAD (Number of Abstract Data types) is the number of user-defined objects used as attributes in a class that are necessary to instantiate an object instance of the class.

NCM (Number of Class Methods in a class) measures the measures available in a class but not in its instances.

NIV (Number of Instance Variables in a class) measures relations of a class with other objects of the program.

NOA (Number Of Ancestors) is the total number of ancestors of a class.

NPA (Number of Public Attributes) counts the number of attributes declared as public in a class.

NPM (Number of Parameters per Method) is the average number of parameters per method in a class.

NRA (Number of Reference Attributes) counts the number of pointers and references used as attributes in a class.

PCM (Percentage of Commented Methods) is the percentage of commented methods.

PDA (Public Data) counts the accesses of public and protected data of a class.

PMR (Percent of Potential Method uses actually Reused) is the percentage of the actual method uses.

PPD (Percentage of Public Data) is the percentage of the public data of a class.

RFC (Response For a Class) is the number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class.

WCS (Weighted Class Size) is the number of ancestors plus the total class method size.

WMC (Weighted Methods per Class) is the sum of the weights of all the class methods.

2.8.2 METHOD METRICS

AMC (Average Method Complexity) is the sum of the cyclomatic complexity of all methods divided by the total number of methods.

AMS (Average Method Size) measures the average size of program methods.

MAG (MAX V(G)) is the maximum cyclomatic complexity of the methods of one class.

MCX (Method Complexity) relates complexity with the number of messages.

2.8.3 COUPLING METRICS

CBO (Coupling Between Objects) counts the number of classes a class is coupled with.

CCP (Class Coupling) measures connections between classes based on the messages they exchange.

CFA (Coupling Factor) is the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance.

2.8.4 INHERITANCE METRICS

AIF (Attribute Inheritance Factor) is the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes.

DIT (Depth of Inheritance Tree) measures the number of ancestors of a class.

FEF (Factoring Effectiveness) is the number of unique methods divided by the total number of methods.

FIN (FAN-IN) is the number of classes from which a class is derived and high values indicate excessive use of multiple inheritance.

HNL (Class Hierarchy Nesting Level) measures the depth in hierarchy that every class is located.

MIF (Method Inheritance Factor) is the ratio of the sum of the inherited methods in all classes to the total number of available methods for all classes.

MRE (Method Reuse metrics) indicate the level of methods reuse.

NMI (Number of Methods Inherited) measures the number of methods a class inherit.

NMO (Number of Methods Overridden) is the number of methods need to be re-declared by the inheriting class.

NOC (Number Of Children) is the total number of children of a class.

PFA (Polymorphism Factor) is the ratio of the actual number of possible different polymorphic situations of a class to the maximum number of possible distinct polymorphic situations for this class.

PMO (Percent of Potential Method uses Overridden) is the percentage of the overridden methods.

RDB (Ratio between Depth and Breadth) is the ratio between the depth and the width of the hierarchy of the classes.

RER (Reuse Ratio) is the ratio of the number of superclasses divided by the total number of classes.

SIX (Specialisation Index) measures the type of specialisation.

SPR (Specialisation Ratio) is the ratio of the number of subclasses divided by the number of superclasses.

2.8.5 SYSTEM METRICS

ADI (Average Depth of Inheritance) is computed by dividing the sum of nesting levels of all classes by the number of classes.

ANA (Average Number of Ancestors) determines the average number of ancestors of all the classes.

APG (Application Granularity) is the total number of objects divided by the total number of function points.

ASC (Association Complexity) measures the complexity of the association structure of a system.

CAN (Category Naming) divides classes into semantically meaningful sets.

CRE (Number of time a Class is Reused) measures the references in a class and the number of the applications that reuse this class.

FDE (Functional Density) is the ratio of LOC to the function points.

NCT (Number of Classes Thrown away) measures the number of times a class is rejected until it is finally accepted.

NOH (Number Of Hierarchies) is the number of distinct hierarchies of the system.

OLE (Object Library Effectiveness) is the ratio of the total number of object reuses divided by the total number of library objects.

PRC (Problem Reports per Class) measures defect reports on this class.

PRO (Percent of Reused Objects Modified) declares the percentage of the reused objects that have been modified.

SRE (System Reuse) declares the percentage of the reuse of classes.

III. Conclusion and Future Work

We conclude that the Object oriented metrics exist and do provide valuable information to object oriented developers and project managers. The SATC has found that a combination of “traditional” metrics and metrics that measure structures unique to object oriented development is most effective. This allows developers to continue to apply metrics that they are familiar with, such as complexity and lines of code to a new development environment. However, now that new concepts and structures are being applied, such inheritance, coupling, cohesion, methods and classes, metrics are needed to evaluate the effectiveness of their application. Metrics such as Weighted Methods per Class, Response for a Class, and Lack of Cohesion are applied to these areas. The application of a hierarchical structure also needs to be evaluated through metrics such as Depth in Tree and Number of Children. At this time there are no clear interpretation guidelines for these metrics although there are guidelines based on common sense and experience.

With the software metrics we can measure software’s overall complexity (including all its components and classes). I’m trying to have sufficient experimental results to prepare a research paper. Also there are metrics for measuring software’s run-time properties and would be worth studying more. The future goal is the definition of a minimum set of metrics applicable to the majority of cases and particular sets of metrics applicable to specific (defined) programming cases and to develop a framework that facilitates measurements of all metrics

References.

[1] Abreu, F. B. e., "The MOOD Metrics Set," presented at ECOOP '95 Workshop on Metrics, 1995.

[2] Abreu, F. B. e. and Melo, W., "Evaluating the Impact of OO Design on Software Quality," presented at Third International Software Metrics Symposium, Berlin, 1996.

[3] Basili, V. R., Briand, L. C., and Melo, W. L., "A Validation of Object Orient Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 21, pp. 751-761, 1996.

[4] Boehm, B. W., "Improving Software Productivity," *IEEE Computer*, pp. 43-57, September 1987.

[5] Briand, L., Emam, K. E., and Morasca, S., "Theoretical and Empirical Validation of Software Metrics," 1995.

[6] Briand, L., Ikonomovski, S., Lounis, H., and Wust, J., "Measuring the Quality of Structured Designs," *Journal of Systems and Software*, vol. 2, pp. 113-120, 1981.

[7] Briand, L. C., Daly, J. W., and Wust, J. K., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91-121, January/February 1999.

[8] Chidamber, S. R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, 1994.

[9] Churcher, N. I. and Shepperd, M. J., "Comments on 'A Metrics Suite for Object-Oriented Design'," *IEEE Transactions on Software Engineering*, vol. 21, pp. 263-5, 1995.

[10] Coad, P., "TogetherSoft Control Center," pp. http://togethersoft.com.

[11] El Emam, K., "A Methodology for Validating Software Product Metrics," National Research Council of Canada, Ottawa, Ontario, Canada NCR/ERC-1076, June 2000 June 2000.

[12] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole Pub Co., 1998.

[13] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, d., *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison Wesley, 1999.

[14] Glasberg, D., Emam, K. E., Melo, W., and Madhavji, N., "Validating Object-Oriented Design Metrics on a Commercial Java Application," National Research Council 44146, September 2000.

[15] Gustafson, D. A. and Prasad, B., "Properties of Software Measures," in *Formal Aspects of Measurement*, T. Denvir, Ed. New York: Springer-Verlag, 1991.

[16] Harrison, R., Counsell, S. J., and Nithi, R. V., "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, pp. 491-496, June 1998.

[17] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

[18] Korson, T. D. and Vaishnavi, V. K., "An Empirical Study of Modularity on Program Modifiability," *Empirical Studies of Programmers*, pp. 168-86, 1986.

[19] Schneidewind, N. F., "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, vol. 18, pp. 410-422, 1992.

[20] Selby, R. W. and Vasili, V. R., "Analyzing Error-Prone Systems Structure," *IEEE Transactions on Software Engineering*, vol. 17, pp. 141-152, 1991.

[21] Stevens, W., Myers, G., and Constantine, L., "Structured Design," *IBM Systems Journal*, vol. 13, pp. 60-73, 1974.

[22] Shyam R. Chidamber, Chris F. Kemerer, A METRICS SUITE FOR OBJECT ORIENTED DESIGN, 1993

[23] Carnegie Mellon School of Computer Science, Object-Oriented Testing & Technical Metrics, PowerPoint Presentation , 2000

[24] Linda H. Rosenberg, Applying and Interpreting Object Oriented Metrics

[25] Jaana Lindroos, Code and Design Metrics for Object-Oriented Systems, 2004

[26 Ralf Reißing, Towards a Model for Object-Oriented Design Measurement

[27] Magiel Bruntink, Testability of Object-Oriented Systems: a Metrics-based Approach, 2003

[28] Aine Mitchell, James F. Power, Toward a definition of run-time object-oriented metrics, 2003

[29] David N. Card, Khaled El Emam, Betsy Scalzo, Measurement of Object-Oriented Software Development Projects, 2001

[30] Dr.R.V.Krishnaiah, BANDA SHIVA PRASAD, Analysis of object Oriented Metrics, International Journal Of Computational Engineering Research (ijceronline.com) Vol. 2 Issue. 5,sept.2012.

BIOGRAPHY

Pooja Arora is a Research Scholar in the School of computing science and engineering Department, Galgotias University. She received M.Tech(CSE) degree in 2008 from USIT, GGSIPU.. Her research interests are Software engineering. Software metrics, software testing and validation.